Recommendations for Evolving Relational Databases

Julien Delplanque, Anne Etien, Nicolas Anquetil, Stéphane Ducasse

Recommendations for Evolving Relational Databases

Julien Delplanque[1,2], Anne Etien[1,2], Nicolas Anquetil[1,2], and Stéphane Ducasse[2,1]

[1] Univ. Lille, CNRS, Centrale Lille, Inria UMR 9189 - CRIStAL
[2] INRIA Lille Nord Europe, Villeneuve d'Ascq, France
{firstname}.{lastname}@inria.fr

Abstract. Relational databases play a central role in many information systems. Their schemas contain structural and behavioral entity descriptions. Databases must continuously be adapted to new requirements of a world in constant change while: (1) relational database management systems (RDBMS) do not allow inconsistencies in the schema; (2) stored procedure bodies are not meta-described in RDBMS such as PostgreSQL that consider their bodies as plain text. As a consequence, evaluating the impact of an evolution of the database schema is cumbersome, being essentially manual. We present a semi-automatic approach based on recommendations that can be compiled into a SQL patch fulfilling RDBMS constraints. To support recommendations, we designed a meta-model for relational databases easing computation of change impact. We performed an experiment to validate the approach by reproducing a real evolution on a database. The results of our experiment show that our approach can set the database in the same state as the one produced by the manual evolution in 75% less time.

Keywords: relational database, meta-model, semi-automatic evolution, impact analysis

1 Introduction

Relational Database (DB) schemas contain structural entity descriptions (*e.g.,* tables and columns), but also sometimes descriptions of behavioral entities such as views (*i.e.,* named `SELECT` queries), stored procedures (*i.e.,* functions written in a programming language), triggers (*i.e.,* entity listening to events happening on a table and reacting to them), etc. Structural and behavioral entities are referencing each others through foreign keys, function calls, or table/column references in queries.

Continuous evolution happens on databases [21] to adapt to new requirements of a world in constant change. When databases evolve, problems are twofold:

Issue 1: Relational database management systems (RDBMS) do not allow schema inconsistencies. The consistency of databases is ensured by the RDBMS *at any moment*. This feature makes the evolution of the database complicated because the database runs during the evolution and continues to ensure its consistency. For other kinds of software, the program is stopped during source code edition. Thus, the program can be temporarily in an inconsistent state.

Issue 2: Stored procedure bodies are not meta-described in RDBMS such as PostgreSQL. Unlike references between tables, columns, constraints or views that are kept

and managed through metadata, stored procedures bodies are considered only as text and existing references they make are not known. This second problem slightly alters the first one as inconsistencies can be introduced but only in stored procedures (dangling references).

For example, to remove a column from a table, different cases occur:

(i) If the column is not referenced, the change can be performed.
(ii) If the column is a primary key and is referenced by a foreign key in another table, the removal is not allowed.
(iii) If the column is referenced in a view, either the change is refused or the view must be dropped. In the latter case, views referencing the one to drop must also be transitively dropped.
(iv) If the column is referenced in a function, change can be performed but an error might arise at execution.

Cases (ii) and (iii) result from the first issue whereas the second issue leads to case (iv). This shows that the consequences of even a small change can be complex to handle, particularly cases (iii) and (iv). Such changes need to be anticipated to comply with the constraints imposed by the RDBMS. Meurice *et al.* studied the history of three applications using databases [14]. They conclude that the impact of renaming or removing a table or a column on programs using the database is not trivial. To ease the evolutions of the database and the management of their impacts on related programs, the authors provide a tool to detect and prevent program inconsistencies under database schema evolution. Their approach has two major drawbacks: First, only a small number of evolutions of the database are taken into account (removing and renaming table or column); second, internal programs stored in behavioral entities such as views or stored procedures are not studied.

In this paper, we propose a tool automating most modifications required after applying a change on a database, similar to a refactoring browser [18, 19]. We do not consider only refactorings [5], which are by definition behavior preserving, but also deal with other evolutions as illustrated by the above example. Thus, we use the term *change* rather than *refactoring*.

We propose an approach based on a meta-model to provide recommendations to database architects. The architects initiate a change and, based on impact analysis, our tool proposes recommendations to the architect. Those recommendations allow the model to reach a consistent state after the change – new changes are induced and new recommendations provided until a stable state is reached. Finally, when the architect accepts the various changes, an analysis is done to generate a patch containing all the SQL queries to perform these changes.

This article is organized as follows. Section 2 sets the context and defines the vocabulary. Section 3 introduces the behavior-aware meta-model used to represent relational databases. Section 4 describes our approach based on impact computation and recommendations to generate SQL evolution patch. It also shows an example of how our approach manages such evolutions illustrated with one evolution operator. Section 5 validates our approach by using our implementation to reproduce an evolution that was performed by an architect on a real database. Section 6 discusses related work. Finally, Section 7 concludes this article by summarizing our results and proposing future work.

2 Setting the context

Before getting into the meta-model and approach explanations, let us set the context in which the approach is designed.

Database schema: The concept of database schema commonly refers to the way data are organized in a database (through tables and referential integrity constraints for relational databases). However, RDBMSs also allows one to define behavior inside the database (*e.g.,* stored procedure), and this behavior might be used to constrain data (*e.g.,* triggers or `CHECK` constraints). Thus, since there is a fine line between the schema as described before and the behavior, in this article when the terms *database schema* or *schema* are used, they refer to both structural and behavioral entities.

Impact of a change: Changing a database will probably affect its structure and behavior. The *impact* of such a change is defined as the set of database entities that potentially need to be adapted for the change to be applied. For example, `RemoveColumn`'s impact set includes constraints applied to the column.

Recommendation: Once the *impact of a change* has been computed, decisions might need to be taken to handle impacted entities, for example dropping views in cascade in the scenario proposed in the introduction. In the context of this paper, we call each of these potential decisions a *recommendation*. For example, if one wants to remove a column, we recommend to remove the `NOT NULL` constraint concerning this column.

Note that Bohnert and Arnold definition of *impact* [1] mixes the set of impacted entities and the actions to be done to fix such entities (in the context of this paper, we call these actions "recommendations"): "Identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change". To avoid confusion, we decided to use a specific word for each part of the definition.

We identified two kinds of constraints involved in a relational database: (1) data constraints are responsible for data consistency. 5 types of such constraints are available: "primary key", "foreign key", "unique", "not-null" and "check". (2) schema constraints are responsible for schema consistency and 3 types of such constraints are available: "a table can have a single primary key", "a column can not have the same constraints applied twice on it" and "foreign key can not reference a column that has no primary key or unique constraint".

Database schema consistency: The RDBMS ensures the consistency of the database schema. This notion of consistency is characterized by the fact that schema constraints are respected and no dangling reference is allowed (except in stored procedure).

Our approach works on a model of the database schema. Using a model allows one to temporarily relax schema constraints and dangling references constraint for the sake of evolution. It allows the developer to focus on changes to be made and not on how to fulfill schema consistency constraints and avoid dangling references at any time.

Operator: An operator represents a change to the database schema. It may impact several entities and require further changes to restore the schema in a consistent state after its application. `RemoveColumn` is an example of operator.

Entity-oriented operator: An entity-oriented operator applies on an element of the model that does not represent a reference. This kind of operator has the particularity to be translatable directly as one or many SQL queries that implement it. An example of such operator is `RemoveColumn`.

Reference-oriented operator: A reference-oriented operator applies on an element of the model representing a reference. RDBMSs do not reify references. Thus, such concepts are implicit and only exist in the source code of DB entities. Because of that, they can not be directly translated as SQL queries. Instead, they need to be converted to entity-oriented operator by interpreting them and generating updated versions of the source code of concerned entities. An example of such operator is `ChangeReferenceTarget`.

3 A Behavior-Aware Meta-Model for Relational Databases

This section presents our meta-model for relational databases. It takes into account both structural and behavioral entities of the database as well as their relationships.

3.1 Meta-Model Objectives

As discussed in the introduction, modifying the structure of a database implies adapting the behavior (*i.e.* program) depending on it. Thus, the development of the meta-model is driven by two objectives:

1. Model the structure and behavior of the database.
2. Ease the computation of entities impacted by a change.

Objective 1 is fulfilled by modeling tables, columns and constraints. We also model behavioral entities such as CRUD[3] queries, views (*i.e.,* named SELECT query stored in the database), stored procedures, and triggers. Objective 2 is fulfilled by *reifying references* between structural and behavioral entities. The details of these modeling choices are given in Section 3.4.

The implementation of the meta-model is available on github[4]. The meta-model is instantiated by analysing meta-data provided by the RDBMS and parsing the source code of entities that are not meta-described. The source code of the meta-data reader[5] and the parser[6] available on github as well.

3.2 Structural Entities

Figure 1 shows the structural part of the meta-model. To ease reading, for this UML diagram and the following, inheritance links have straight corners while other links are rounded; classes modeling structural entities are red (such as Table); classes modeling behavioral entities are orange (such as StoredProcedure); and classes modeling references are white.

A StructuralEntity defines the structure of data held by the database or defining constraints applied on these data (*e.g.,* Table, Column, Referential integrity constraint, etc.).

[3] Create Read Update Delete query in SQL: INSERT, SELECT, UPDATE, DELETE.

[4] https://github.com/juliendelplanque/FAMIXNGSQL

[5] https://github.com/olivierauverlot/PgMetadata

[6] https://github.com/juliendelplanque/PostgreSQLParser

The containment relation between Table and Column is modeled through ColumnsContainer which is an abstract entity. This entity also has sub-classes in the behavioral part of the meta-model (see Section 3.3). A Column has a type. This relation is modeled through a TypeReference. A Column can also be subject to Constraints. Depending on whether a Constraint concerns a single or multiple columns, it inherits from, respectively, ColumnConstraint or TableConstraint. Six concrete constraints inherit from Constraint: PrimaryKey, ForeignKey, Unique, Check (a developer-defined constraint, described by a boolean expression), NotNull, and Default (a default value assigned when no value is explicitly provided, it can be a literal value or an expression to compute). Note that Check and Default constraints also inherit from BehavioralEntity because they contain source code.

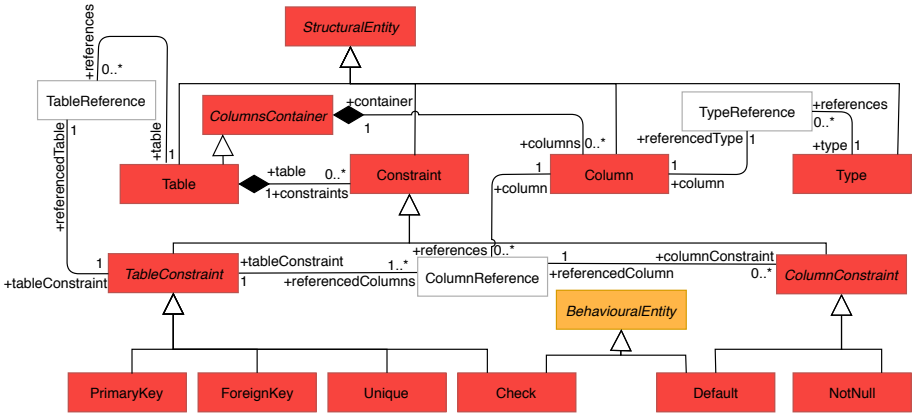

Fig. 1. Structural entities of the meta-model.

3.3 Behavioral Entities

A behavioral entity is an entity holding behavior that may interact with StructuralEntities. Figure 2 shows the behavioral part of the meta-model. The main entities are as follows.

View is a named entity holding a SELECT query. StoredProcedure is an entity holding developer-defined behavior which includes queries and calls to other StoredProcedure. A StoredProcedure contains Parameter(s) and LocalVariable(s). These entities can be referenced in clauses of queries that are contained in StoredProcedures or Views. Trigger represents actions happening in response to event on a table (*e.g.,* row inserted, updated or deleted). CRUDQuery(ies) contain multiple clauses depending on the query. For the sake of readability, we did not include the clause classes in the diagram. In a nutshell, the containment relation between CRUD queries and clauses are: SelectQuery contains With, Select, From, Where, Join, Union, Intersect, Except, GroupBy, OrderBy, Having, Limit, Offset, Fetch clauses. InsertQuery contains With, Into, Returning clauses. UpdateQuery contains With, Update, Set, From, Where, Returning clauses. DeleteQuery contains With,

Delete, From, Where, Returning clauses. Each clause holds some References to structural or behavioral entities. The references made to structural or behavioral entities from clauses are detailed in Section 3.4. DerivedTable is an anonymous query usually used in another query but can also appear in the body of a StoredProcedure.

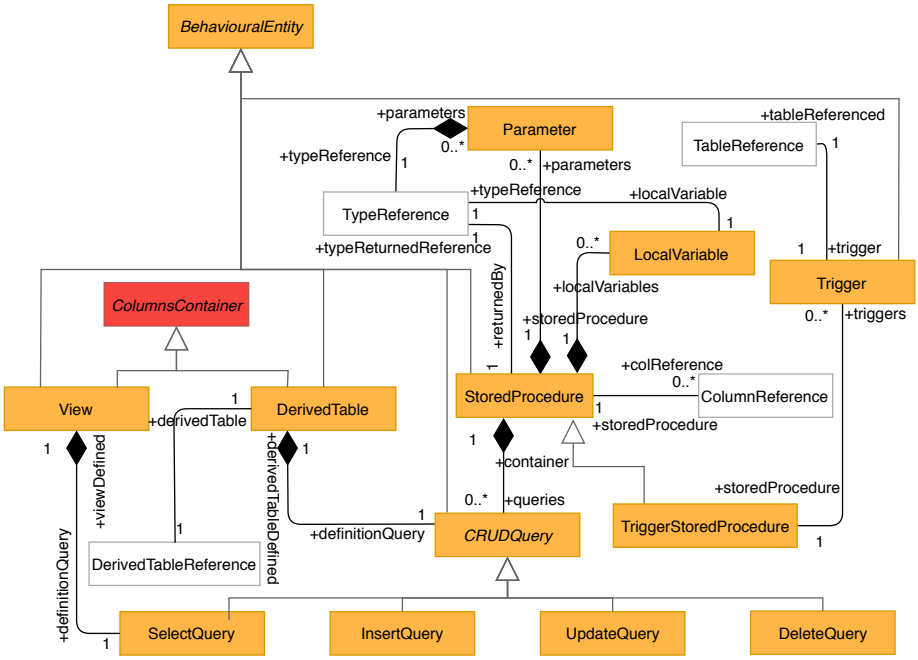

Fig. 2. Behavioral entities of the meta-model.

3.4 References

The third and last part of the meta-model represents links between entities. It allows one to track relations between behavioral and structural entities. To simplify the approach, all references have been reified. For example, a column is thus referenced through a ColumnReference, a local variable through a LocalVariableReference and a stored procedure through a StoredProcedureCall.

4 Description of the Approach

To evolve a database, the database architect formulates changes on some of its entities. These changes impact other entities that, in turn, need to evolve to maintain the database

in a consistent state. To handle evolutions induced by the initial changes, we developed a 3-step approach. The implementation of this approach is available on github[7].

A. *Impact computation*: The set of *impacted* entities is computed from the change. The next step treats impacted entities one by one.
B. *Recommendations selection*: Second, depending on the change, and the *impacted* entity, our approach computes a set of *recommendations*. These *recommendations* are presented to the database architect that chooses one when several are proposed. This introduces new changes that will have new impacts. Steps A. and B. are recursively applied until all the *impacts* have been managed.
C. *Compiling operators as a valid SQL patch*: Finally, all operators (the recommendations chosen by the architect) are converted as a set of SQL queries that can be run by the RDBMS. The set of SQL queries is used to migrate the database to a state in which the initial architect's change has been applied.

4.1 Impact computation

To compute the entities potentially affected by a change, one needs to collect all the entities referencing this changed entity. For example, if a Column is subject to a modification, our approach identifies the *impacted* entities by gathering all the ColumnReferences concerning this column. The *impact* of the change corresponds to the sources of all ColumnReferences since they can potentially be affected by the modification.

4.2 Recommendations selection

For each operator, the set of impacted entities is split into disjoint sub-sets called *categories*. For each of these *categories*, one or several *recommendations* are available. We determinated those recommendations by analysing how to handle them according to database schema constraints.

The output of step 4.1 combined with this step (4.2) is a tree of operators where the root is the change initiated by the architect and, each other node corresponds to an operator chosen among *recommendations*.

4.3 Compiling operators as a valid SQL patch

Once all the *impact sets* have been considered and *recommendations* chosen, our approach generates a SQL patch. This patch includes queries belonging to the SQL data definition language (DDL). These queries enable migrating the database from its original state to a state where the initial operator and all induced operators have been applied.

We stress that, during the execution of any operator of the patch, the RDBMS cannot be in inconsistent state. This constraint is fundamentally different from source code refactoring where the state of the program can be temporarily inconsistent. Therefore, each operator must lead the database to a state complying with schema consistency

[7] https://github.com/juliendelplanque/DBEvolution

constraints. Else the RDBMS will forbid the execution of the SQL patch. For this purpose, the tree of operators resulting from the previous step has to be transformed into a sequence of SQL queries.

The tree resulting from the step described in section 4.2 is composed of operators on references. However, DDL queries only deal with entities. Thus, reference-oriented operators are transformed into entity-oriented operators. As the RDBMS does not allow inconsistencies, operators concerning a given behavioral entity of the database are aggregated into a single operator per view and per stored procedure. This aggregation is performed in two steps: 1. all reference-oriented operators are grouped according to the entity to which belongs to the source code in which the reference appears, and 2. for each group of reference-oriented operators, we create the new version of the source code for this entity. To do so, we iterate the list of reference-oriented operators and update the part of the source code corresponding to the reference to make it reflect the change implemented by the operator. Once the iteration is complete, a new version of the source code has been built with no more dangling reference.

Those entity-oriented operators are ordered to comply with RDBMS constraints of consistency and serialized as SQL queries. Technical details related to this serialization are not provided in this paper because of space limitation.

4.4 Example

To explain the proposed process, let us take a small example. Consider the simple database shown in Figure 3. In this database, there are two tables, t1 with two columns t1.b, t1.c and t2 with column t2.e. Additionally, one stored procedure s() and three views v1, v2 and v3 are present. On this figure, dependencies between entities are modeled with arrows. These dependencies arrows are a generalization over the various kinds of reference entities of the meta-model. For example, the arrow between s() and t1 is an instance of TableReference and the arrow between s() and b is an instance of ColumnReference. Views and functions have source code displayed inside their box. In this source code, a reference to another entity of the database is underlined.

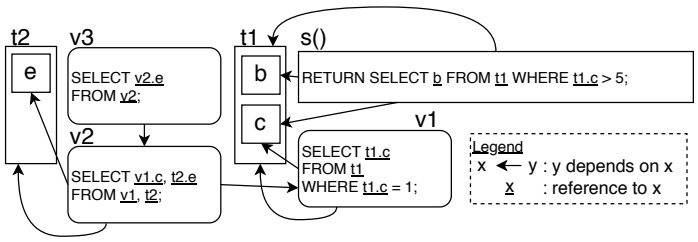

Fig. 3. Example database.

The architect wants to rename the column c of table t1 as d.

Impact computation. First, we compute the impact of this change. Column c of table t1 is referenced three times: *(i)* in the WHERE clause of the SELECT query of the stored

procedure s(); *(ii)* in the WHERE clause of the query defining view v1; and *(iii)* in the SELECT clause of the query defining view v1. Each of these clauses is added in the *impact* of renaming t1.c as t1.d.

Recommendations selection. For each of the three *impacted entities*, *recommendations* are produced. For the WHERE clause of the stored procedure s(), the recommendation is to replace the reference to column t1.c with a new one corresponding to t1.d. The result of replacing this reference will be the following source code: RETURN SELECT b FROM t1 WHERE t1.d $>$ 5;. From this operator, the impact is computed but is empty which stops the recursive process.

The recommendation concerning the WHERE clause of v1 is the same: replacing the reference to t1.c by a reference to t1.d. Again, there is no further impact for this operator.

For the reference to t1.c in the SELECT clause of view v1, two recommendations are proposed to the architect: either aliasing the column and replacing the reference (*i.e.,* replacing SELECT t1.c by SELECT t1.d AS c) or replacing the reference (*i.e.,* replacing SELECT t1.c by SELECT t1.d). In the latter case, the column c in view v1 becomes d; it is no longer possible to refer to v1.c. Consequently, the second recommendation leads to rename column v1.c. If the architect choose to replace the reference without aliasing, the recursive process continues: new impacts need to be computed and new changes to be performed. The SELECT clause of view v2 is impacted. Two recommendations are again provided: either aliasing the column and replacing the reference or just replacing the reference. In this case, the architect chooses to alias the column and replace the reference. Thus, the rest of the database can continue to refer to column c of view v2. Figure 4 illustrates this step.

Compiling operators as a valid SQL patch. Figure 5 illustrates the patch generation step. References-oriented operators resulting from the recommendations are transformed into entity-oriented operators. For this purpose, operators concerning the same sourced entity are aggregated. Operators (3) and (4) concern the same sourced entity, v1. They are thus aggregated into ModifyViewQuery(v1). At the end, there is a single operator per entity to be modified.

The resulting list of operators is ordered and converted to a SQL patch.

5 Experiment

Our university department uses an information system to manage its members, teams, thematic groups, etc. with 95 tables, 63 views, 109 stored procedures and 20 triggers.

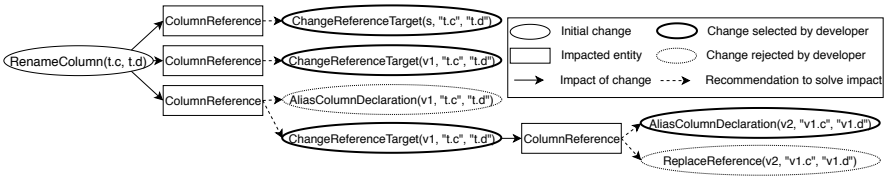

Fig. 4. Recommendations selection.

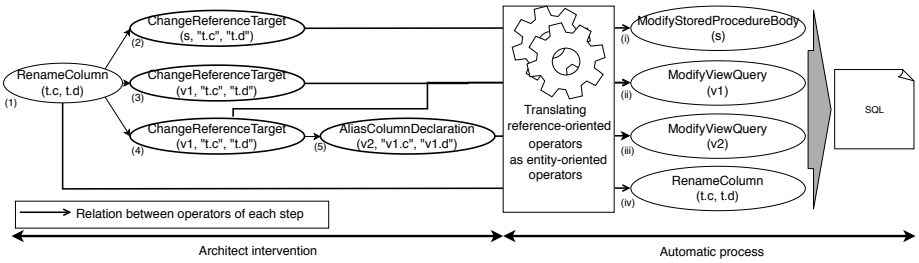

Fig. 5. Compiling operators as a valid SQL patch.

This information system is developed by a database architect. Before each migration, he prepares a road map containing, in natural language, the list of operators initially planned for the migration. We observed that these road maps are not complete or accurate [4]. Following a long manual process, the architect writes a SQL patch (*i.e.,* a text file containing queries) to migrate from one version of the database to the next one.

The architect gave us access to these patches to do a post-mortem analysis of the DB evolutions. One of the patches implements the renaming of a column belonging to a table that is central to the DB. This is interesting because it is a non-trivial evolution.

We had the opportunity to record the architect's screen during this migration [4]. We observed that the architect used a trial-and-error process to find dependencies between entities of the database. He implements part of the patch and runs it in a transaction that is always rolled back. When the patch fails in between, the architect uses the gained knowledge to correct the SQL patch. Using this methodology, the architect built incrementally the SQL patch implementing the patch during approximately 1 hour. The patch is ~ 200 LOC and is composed of 19 SQL statements. To validate our approach, we regenerate this SQL patch with our tool but without the architect's expertise. Then, we compare our resulting database with the one obtained by the architect.

5.1 Experimental Protocol

The goals of the experiment are multiple: *(i)* to illustrate on a concrete case the generation of a SQL patch; *(ii)* to compare the database resulting from our approach with the one originally written by the architect; and *(iii)* to estimate the time required to generate a SQL patch as compared to the manual generation.

Based on the road map written by the architect and the comments in the patch we extracted the operators initiated by the architect during this migration. A discussion with the architect allowed us to validate the list of initial operators: RenameColumn(person.uid, login), RemoveFunction(key_for_uid(varchar)), RemoveFunction(is_responsible_of(int4)), RemoveFunction(is_responsible_of(int4,int4)), RenameFunction(uid(integer), login(integer)), RenameLocalVariable(login.uidperson, login.loginperson), RemoveView(test_member_view). Details on these operators can be found at: https://hal.inria.fr/hal-02504949v1.

The experiment consists in choosing these operators in our tool and following the *recommendations* it proposes. Potentially several *recommendations* might be proposed,

particularly as whether to create aliases in some referencing queries or to rename various columns in cascade (see example in Section 4.4). The architect told us that, as a rule, he preferred to avoid using aliases and renamed the columns. These were the only decision we had to do during the experiment.

We finished the experiment by executing the SQL patch generated by our tool on an empty (no data) copy of the database. Note that having no data in the database to test the patch might be a problem for operators modifying data (*e.g.,* changing the type of a column implies converting data to the new type). However, in the case of our experiment no operator modifies data stored in the database. First, we checked whether the generated patch ran without errors. Second, we compared the state of the database after the architect's migration and ours. For this, we generated a dump of the SQL schema of both databases and compared these two dumps using a textual diff tool. Third, we also considered the time we spent on our migration and the one used by the architect when he did his.

5.2 Results

We entered the seven operators listed previously in our tool and let it guide us through the decision process to generate the SQL migration patch.

Fifteen decisions were taken to choose among the proposed recommendations. They all concerned the renaming or aliasing of column references. From this process, the tool generated a SQL patch of ~ 270 LOC and 27 SQL statements.

To answer the goals of the experiment listed previously: *(i)* The generated SQL patch was successfully applied on the database. *(ii)* The diff of the two databases (one being the result of the hand-written patch and the other being the result of the generated patch) showed a single difference: a comment in one function is modified in the hand-written version. Such changes are not taken into account by our approach. *(iii)* Encoding the list of changes and taking decisions took approximately 15 minutes. This corresponds to about 25% of the time necessary to the architect who has a very good knowledge of his database to obtain the same result.

5.3 Discussion

Validating tools predicting the impact of a software change is not easy. Evidence of that claim can be found in Lehnert's meta-review [10]. On the 18 approaches reviewed by Lehnert using either call graphs or program dependency graph techniques, only six have experimental results about the size of the system, time, precision and recall. And only one of these has results on all the metrics together.

Accessing industrial databases with their evolutions is more difficult than accessing source code. Since databases are usually at the core of company business, companies are reluctant to provide their schema. The database schema evolutions are not systematically recorded in tools such as version control systems (VCS) probably because the integration between relational database and VCS is poor. Finding database administrators willing to devote some time to our experiment can also be challenging.

It is also possible to analyze the co-evolution between the source code of a database and the source code of its clients. Analyzing only the behavior inside the database has

the advantage that the precision is better as queries are usually not built dynamically. When queries are built dynamically via string concatenation, it is hard to determinate what query is executed in the end. However, it is possible to build query dynamically from inside the database (via `PERFORM` query). We do not handle these kinds of query at the moment but it would be possible to use an approach similar to Meurice et al. approach [14].

Note that our approach has been applied on AppSI database but is does not rely on AppSI specificities. DBEvolution relies on the meta-model and operators definitions to provide recommendations for a given change. We can import other databases as model in our tool. For example, we were able to load Liquidfeedback database schema[8] in our tool and we can use DBEvolution on it to get recommendations.

6 Related Work

Our work needs to be compared to *impact analysis* and *database schema evolution* research fields.

Impact Analysis Since the first paper introducing Impact Analysis by Bohnert and Arnold [1], the research field has been widely investigated by the scientific community. Meta-analyses on this topic exist, *e.g.,* Lehnert did a review of software change impact analysis in 2011 [10]. We focus on work adapting impact analysis techniques to relational databases as discussed below.

Karahasanovic and Sjøberg proposed a tool, called SEMT, to find impacts of object-database schema changes on applications [9]. Their tool allows one to identify and visualize the impact. It uses an improved version of the transitive closure algorithm. It also provides a language to graphically walk the impact graph.

Gardikiotis and Malevris [6] proposes an approach to estimate the impact of a database schema change on the operability of a web application. To achieve that, they proposed a tool named DaSIAn (Database Schema Impact Analyzer) based on their approach. This tool finds CRUD queries and stored procedures affected by a change on the database schema. The authors also presented an approach assessing impact on client applications from schema changes [7]. They used this approach to assess both affected source code statements and affected test suites in the application using the database after a change in the database.

Maul *et al.,* [13] created a static analysis technique to assess the impact of changing a relational database on its object-oriented software clients. They implemented Schema Update Impact Tool Environment (SUITE) which takes the source code of the application using the database and a model of the database schema as input. Then, they queried this model to to find out the part of the source code application impacted when modifying an entity of the database.

Nagy *et al.,* [15] compared two methods for computing dependencies between stored procedures and tables in a database: One using Static Execute After/Before relations [8] and the other analysing CRUD queries and schema to find database access and propagate this dependency at the stored procedure level. The authors concluded that the two

[8] https://liquidfeedback.org

approaches provide different results and should thus be used together to assess dependencies safely.

Liu *et al.,* [11, 12], proposed a graph called attribute dependency graph to identify dependencies between columns in a database and parts of client software source code using it. They evaluated their approach on 3 databases and their clients written in PHP. Their tool presents to the architect an overview of a change impact as a graph.

Similarly to approaches covered by Lehnert meta-analysis, the validations for impact analysis on databases are usually quite weak because it is a difficult task. To position our approach, it uses static analysis to determine the impact of a change on an entity. This information is directly available in our model because we reify the references between entities. As explained previously, our approach considers that if you change an entity, all entities referencing it are potentially impacted. That set of impacted entities is decomposed into categories and a recommendation is provided for each of them.

Recommendations for Relational Database Schema Evolution Sjøberg's work [20] quantifies schema evolution. They studied the evolution of a relational database and its application forming a health management system during 18 months. To do so they used "the Thesaurus" tool which analyses how many screens, actions and queries may be affected by a potential schema change. This tool does not propose recommendations to users but rather shows code locations to be manually modified. Their results suggest that change management tools are needed to handle this evolution.

Curino *et al.,* [3, 2] proposed PRISM, a tool suite allowing one to predict and evaluate schema modification. PRISM also propose database migration feature through rewriting queries and application to take into account the modification. To do so, they provide a language to express schema modification operators, automatic data migration support and documentation of changes applied on the database. They evaluated their approach and tool on Wikimedia showing it is efficient. In PRISM approach, the operators are limited to modification on structural entities of the database, whereas our approach also deals with change on behavioral entities.

Papastefanatos *et al.,* [16, 17] developed Hecataeus, a tool representing the database structural entities, the queries and the views, as a uniform directed graph. Hecataeus allows user to create an arbitrary change and to simulate it to predict its impact. From this perspective, it is close to the aim of our tool. The main difference is that our approach ensures no inconsistency is created at some point during database evolution. It is not clear how Hecataeus addresses this problem in these papers.

Meurice *et al.,* [14] presented a tool-supported approach that can analyze how the client source code and database schema co-evolved in the past and to simulate a database change to determine client source code locations that would be affected by the change. Additionally, the authors provide strategies (recommendations and warnings) for facing database schema change. Their recommendations describe how to modify client program source code depending on the change performed on the database. The approach presented has been evaluated by comparing historical evolution of a database and its client application with recommendations provided by their approach. From the historical analysis the authors observed that the task of manually propagating database schema change to client software is not trivial. Some schema changes required multiple versions of the software application to be fully propagated. Others were never fully propagated. We argue that, according to what we observed in previous research [4] and

the research made in this article, propagating structural change to behavior entities of the database is a hard task as well.

Compared to previous approaches, DBEvolution brings as a novelty that any entity can be subject to an evolution operator. In particular, stored procedures can be modified and DBEvolution will provide recommandations for the modification. The other way around, modifying a structural entity will provide recommandations to accomodate stored procedures with the change. Such capability is absent from above approaches.

7 Conclusion

We have developed an approach to manage relational database evolution. This approach addresses the two main constraints that a RDBMS sets: 1. *no schema inconsistency is allowed during the evolution* and 2. *stored procedures bodies are not described by metadata*. Addressing these problems allowed us to provide three main contributions: i. a meta-model for relational databases easing the computation of the impact of a change, ii. a semi-automatic approach to evolve a database while managing the impact, and iii. an experiment to assess that our approach can reproduce a change that happened on a database used by a real project with a gain of 75% of the time. These results show that this approach is promising to build the future of relational databases integrated development environments.

Our future works are threefold. First, we would like to extend the set of operators supported by our implementation. More specifically, we want higher-level operators such as: *historize column* which will modify the database schema to keep track of the history of the values of a column through the database life.

Second, the evolution has been reproduced by us which might bias our results in terms of time to implement a change. Indeed, as we have little knowledge on the DB, it is possible that an expert using our tool would be faster than us. Thus, we would like to do another experiment where we compare the performances of an architect using our tool with the performances of an architect using typical tools to implement an evolution.

Finally, some operators will require to transform or move data stored in the database (for example moving a column from a table to another). We plan to support such operators in our methodology by generating CRUD queries in addition to the DDL queries already generated by the operators.

References

1. Arnold, R.S., Bohnert, S.: Software change impact analysis. IEEE Computer Society Press (1996)
2. Curino, C., Moon, H.J., Zaniolo, C.: Automating database schema evolution in information system upgrades. In: Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades. p. 5. ACM (2009)
3. Curino, C.A., Moon, H.J., Zaniolo, C.: Graceful database schema evolution: the prism workbench. Proceedings of the VLDB Endowment **1**(1), 761–772 (2008)
4. Delplanque, J., Etien, A., Anquetil, N., Auverlot, O.: Relational database schema evolution: An industrial case study. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME) (2018). https://doi.org/10.1109/ICSME.2018.00073, http://rmod.inria.fr/archives/papers/Delp18c-ICSME-DatabaseSchemaEvolution.pdf

5. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)
6. Gardikiotis, S.K., Malevris, N.: DaSIAn: A Tool for Estimating the Impact of Database Schema Modifications on WEB Applications. In: Computer Systems and Applications, 2006. IEEE International Conference on. pp. 188–195. IEEE (2006)
7. Gardikiotis, S.K., Malevris, N.: A two-folded impact analysis of schema changes on database applications. International Journal of Automation and Computing **6**(2), 109–123 (2009)
8. Jász, J., Beszédes, Á., Gyimóthy, T., Rajlich, V.: Static execute after/before as a replacement of traditional software dependencies. In: 2008 IEEE International Conference on Software Maintenance. pp. 137–146. IEEE (2008)
9. Karahasanovic, A., Sjoberg, D.I.: Visualizing impacts of database schema changes-a controlled experiment. In: Human-Centric Computing Languages and Environments, 2001. Proceedings IEEE Symposia on. pp. 358–365. IEEE (2001)
10. Lehnert, S.: A review of software change impact analysis. Ilmenau University of Technology p. 39 (2011)
11. Liu, K., Tan, H.B.K., Chen, X.: Extraction of attribute dependency graph from database applications. In: Software Engineering Conference (APSEC), 2011 18th Asia Pacific. pp. 138–145. IEEE (2011)
12. Liu, K., Tan, H.B.K., Chen, X.: Aiding maintenance of database applications through extracting attribute dependency graph. Journal of Database Management **24**(1), 20–35 (2013)
13. Maule, A., Emmerich, W., Rosenblum, D.: Impact analysis of database schema changes. In: Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on. pp. 451–460. IEEE (2008)
14. Meurice, L., Nagy, C., Cleve, A.: Detecting and preventing program inconsistencies under database schema evolution. In: Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on. pp. 262–273. IEEE (2016)
15. Nagy, C., Pantos, J., Gergely, T., Besz'edes, A.: Towards a safe method for computing dependencies in database-intensive systems. In: Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on. pp. 166–175. IEEE (2010)
16. Papastefanatos, G., Anagnostou, F., Vassiliou, Y., Vassiliadis, P.: Hecataeus: A what-if analysis tool for database schema evolution. In: Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on. pp. 326–328. IEEE (2008)
17. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Vassiliou, Y.: Hecataeus: Regulating schema evolution. In: Data Engineering (ICDE), 2010 IEEE 26th International Conference on. pp. 1181–1184. IEEE (2010)
18. Roberts, D., Brant, J., Johnson, R.E., Opdyke, B.: An automated refactoring tool. In: Proceedings of ICAST '96, Chicago, IL (Apr 1996)
19. Roberts, D.B.: Practical Analysis for Refactoring. Ph.D. thesis, University of Illinois (1999), http://historical.ncstrl.org/tr/pdf/uiuc_cs/UIUCDCS-R-99-2092.pdf
20. Sjøberg, D.: Quantifying schema evolution. Information and Software Technology **35**(1), 35–44 (1993)
21. Skoulis, I., Vassiliadis, P., Zarras, A.: Open-source databases: Within, outside, or beyond lehman's laws of software evolution? In: International Conference on Advanced Information Systems Engineering. pp. 379–393. Springer (2014)